# CS 401

## Divide and Conquer

Xiaorui Sun

# Counting inversions

# Counting Inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j inverted if $i < j$, but $a_i > a_j$.

*Songs*

| | | | | | |
|---|---|---|---|---|---|
| Me | 1 | 2 | 3 | 4 | 5 |
| You | 1 | 3 | 4 | 2 | 5 |

Inversions

3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs i and j.

# Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 4 | 8 | 10 | 2 | 6 | 9 | 12 | 11 | 3 | 7 |

Divide: O(1).

| | | | | | |
|---|---|---|---|---|---|
| 1 | 5 | 4 | 8 | 10 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| 6 | 9 | 12 | 11 | 3 | 7 |

Conquer: 2T(n / 2)

5 blue-blue inversions

5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

# Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where $a_i$ and $a_j$ are in different halves, and return sum of three quantities.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 4 | 8 | 10 | 2 | 6 | 9 | 12 | 11 | 3 | 7 |

Divide: O(1).

| | | | | | |
|---|---|---|---|---|---|
| 1 | 5 | 4 | 8 | 10 | 2 |

5 blue-blue inversions

| | | | | | |
|---|---|---|---|---|---|
| 6 | 9 | 12 | 11 | 3 | 7 |

8 green-green inversions

Conquer: 2T(n / 2)

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Enumerate all blue-green pairs takes $O(n^2)$ time

# Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where $a_i$ and $a_j$ are in different halves.

| | | | | | |
|---|---|---|---|---|---|
| 3 | 7 | 10 | 14 | 18 | 19 |

| | | | | | |
|---|---|---|---|---|---|
| 2 | 11 | 16 | 17 | 23 | 25 |
| 6 | 3 | 2 | 2 | 0 | 0 |

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 7 | 10 | 11 | 14 | 16 | 17 | 18 | 19 | 23 | 25 |

Count: O(n)

Combine:

- Sort two halves.
- Count inversions where $a_i$ and $a_j$ are in different halves.

$O(n(\log n)^2)$ time

Sort: O(n log n)

# Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where $a_i$ and $a_j$ are in different halves.
- Merge two sorted halves into sorted whole.

to maintain sorted invariant

| | | | | | |
|---|---|---|---|---|---|
| 3 | 7 | 10 | 14 | 18 | 19 |

| | | | | | |
|---|---|---|---|---|---|
| 2 | 11 | 16 | 17 | 23 | 25 |
| 6 | 3 | 2 | 2 | 0 | 0 |

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: O(n)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 7 | 10 | 11 | 14 | 16 | 17 | 18 | 19 | 23 | 25 |

Merge: O(n)

$$T(n) \le T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + O(n) \quad \Rightarrow \quad \mathrm{T}(n) = O(n \log n)$$

# Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.

Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (r_A, A) ← Sort-and-Count(A)
   (r_B, B) ← Sort-and-Count(B)
   (r, L) ← Merge-and-Count(A, B)

   return r = r_A + r_B + r and the sorted list L
}
```

# Lesson

Sometimes, it is useful to redefine the problem to make the recursion work

In the counting inversions problem

- The merge step becomes easier if two halves are sorted
- So, we redefine the problem (as well as the subproblems) as finding the number of inversions and sorting the input

# Binary Search

# Search Algorithms

<u>Search</u>: Given an element and an array, is the element in the array?

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 53 | 72 | 14 | 97 | 33 | 93 | 51 | 6 | 96 | 10 | 84 | 45 | 95 | 64 | 25 |

Search for 96: Yes

Search for 11: No

Naïve algorithm: Sequential search

- O(n) running time

Can we do better?

# Let us try divide and conquer

Divide and conquer

- Divide: separate list into two pieces.
- Conquer: recursively find the required element in each half.
- Combine: return yes if any subproblem returns yes, otherwise, no

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 53 | 72 | 14 | 97 | 33 | 93 | 51 | 6 | 96 | 10 | 84 | 45 | 95 | 64 | 25 |

Divide: $O(1)$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 53 | 72 | 14 | 97 | 33 | 93 | 51 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 96 | 10 | 84 | 45 | 95 | 64 | 25 |

Conquer: $2T(n / 2)$

Search for 96: No Yes $\Longrightarrow$ Yes

Search for 11: No No $\Longrightarrow$ No

Combine: $O(1)$

$$T(n) = 2T(n/2) + O(1)$$

# Let us try divide and conquer

Divide and conquer

- Divide: separate list into two pieces.
- Conquer: recursively find the required element in each half.
- Combine: return yes if any subproblem returns yes, otherwise, no

| 53 | | | | | | 93 | 51 | 6 | 96 | 10 | 84 | 45 | 95 | 64 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Divide: $O(1)$.

| 53 | | | | | | 51 | 6 | | 96 | 10 | 84 | 45 | 95 | 64 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Conquer: $2T(n / 2)$

Searc

Searc

Yes $\Longrightarrow$ Yes

No $\Longrightarrow$ No

Combine: $O(1)$

**Not better than sequential search** → Overall: $O(n)$

**If no additional condition, O(n) is the best to hope**

# Let us try divide and conquer

Assume the array is sorted

- Divide: separate list into two pieces.
- Conquer: r
- Combine: return yes if any subproblem returns yes, otherwise, no

Key point: avoid recursively solve both

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |

Divide: $O(1)$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 64 | 72 | 84 | 93 | 95 | 96 | 97 |

Conquer: $T(n / 2)$

Combine: $O(1)$

Search for 96: No need to recurse on the first half
- Largest number in the first half < 96

Search for 11: No need to recurse on the second half
- Smallest number in the second half > 11

O(1) time to decide which subproblem to solve!

# Binary Search

Invariant. Algorithm maintains `A[low]` ≤ `key` ≤ `A[high]`.

$\texttt{BinarySearch}(A, low, high, key)$

$\texttt{if}\ \ high < low\texttt{:}$

$\quad\texttt{return}\ \ No$

$mid \leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$

$\texttt{if}\ \ key = A[mid]\texttt{:}$

$\quad\texttt{return}\ \ mid$

$\texttt{else if}\ \ key < A[mid]\texttt{:}$

$\quad\texttt{return BinarySearch}(A, low, mid - 1, key)$

$\texttt{else:}$

$\quad\texttt{return BinarySearch}(A, mid + 1, high, key)$

# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

Ex. Binary search for 33.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| **low** | | | | | | | | | | | | | | **high** |

# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

**Ex.** Binary search for 33.

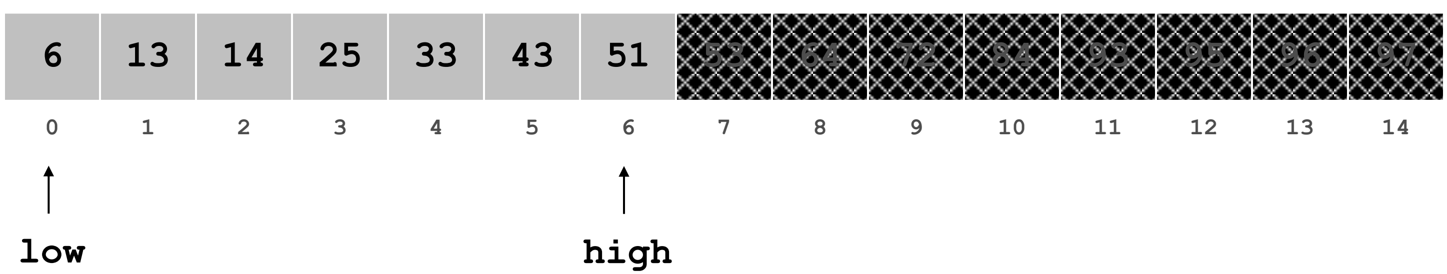
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| low | | | | | | | mid | | | | | | | high |

# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

**Ex.** Binary search for 33.

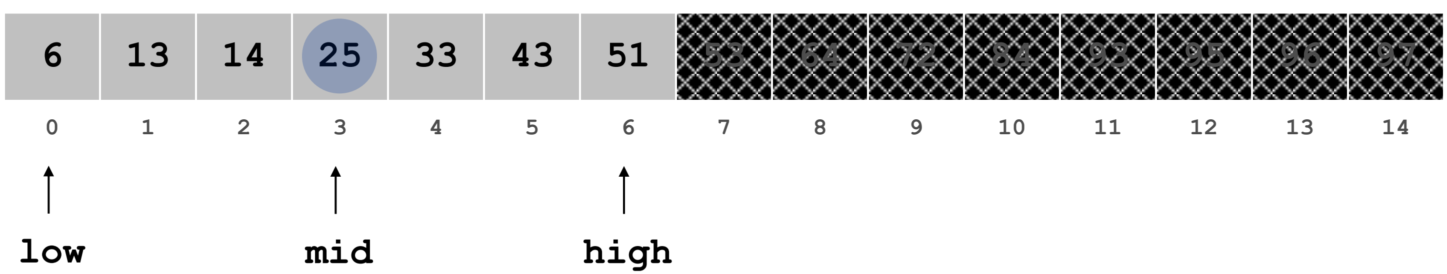


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

**Ex.** Binary search for 33.

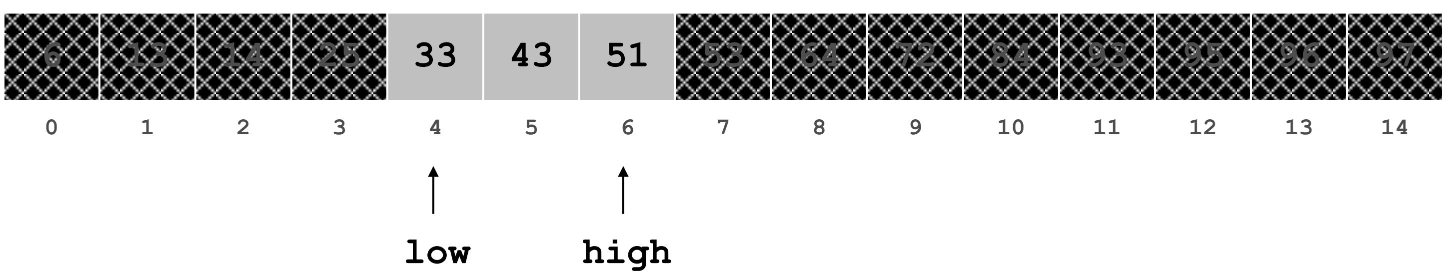


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` ≤ `key` ≤ `A[high]`.

Ex. Binary search for 33.

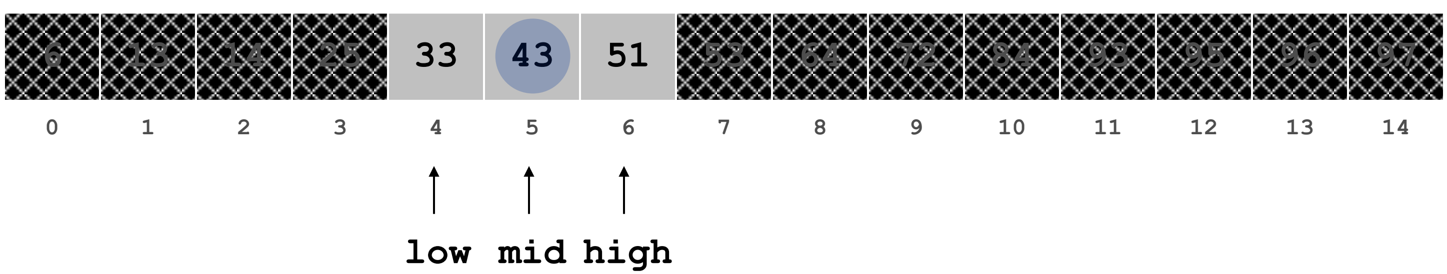


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

Ex. Binary search for 33.

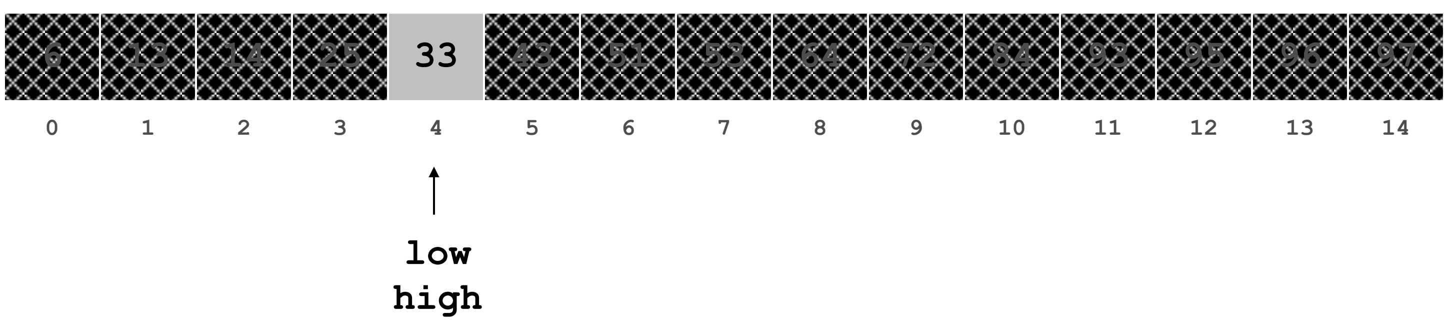


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\le$ `key` $\le$ `A[high]`.

Ex. Binary search for 33.

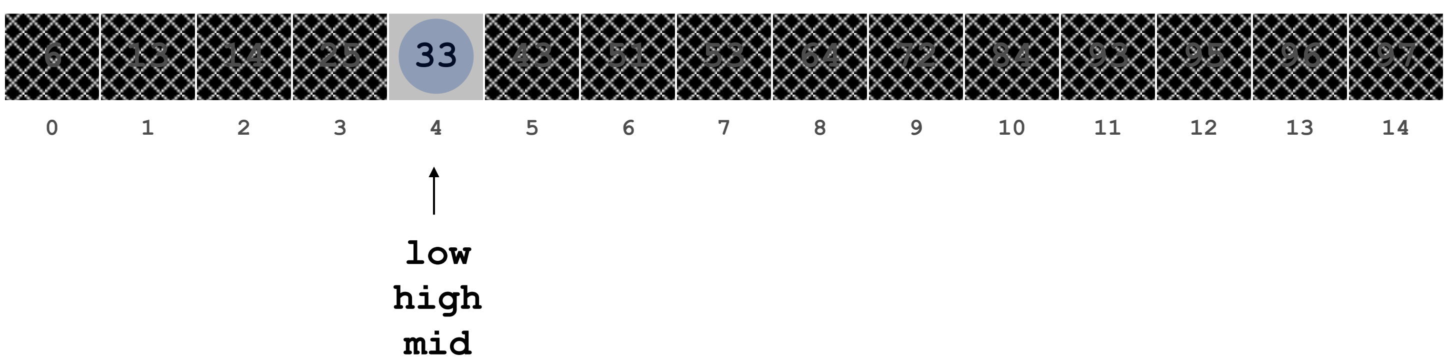


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

**Ex.** Binary search for 33.

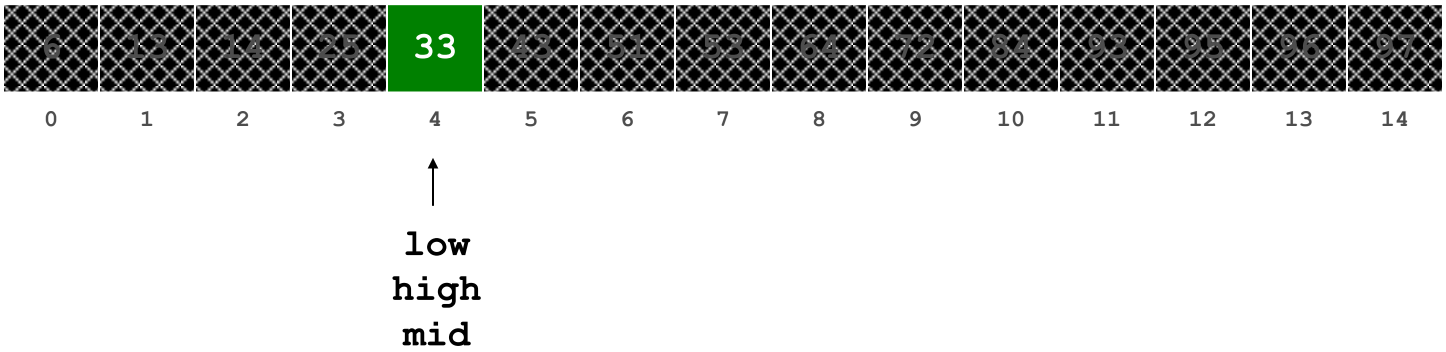


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

Ex. Binary search for 33.



# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains `A[low]` ≤ `key` ≤ `A[high]`.

**Ex.** Binary search for 47.

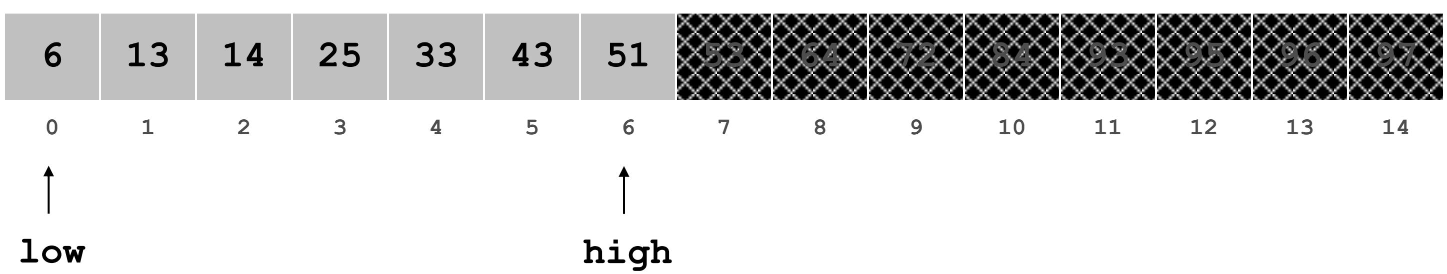
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| ↑ low | | | | | | | ↑ mid | | | | | | | ↑ high |

# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

Ex. Binary search for 47.

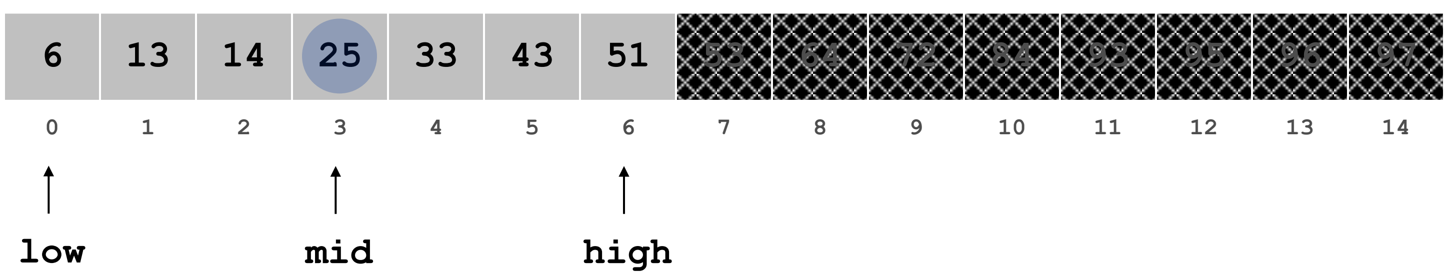


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

Ex. Binary search for 47.

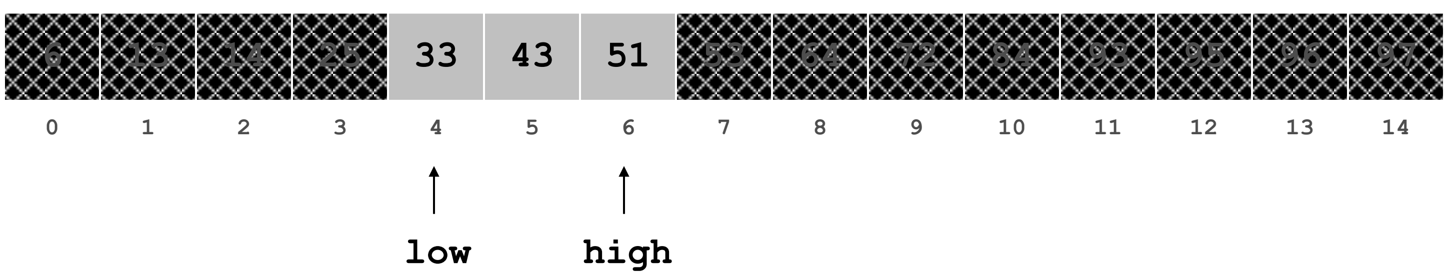


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

Ex. Binary search for 47.

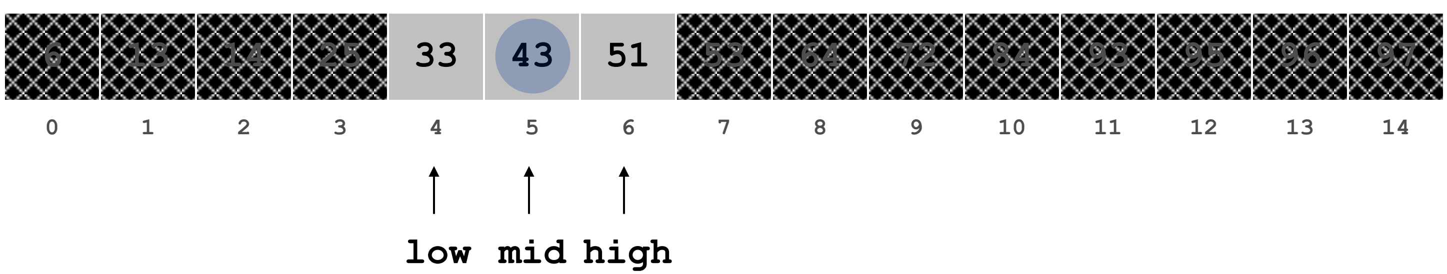


# Binary Search

Binary search. Given key and sorted array A[], find index i such that A[i] = key, or report that no such index exists.

Invariant. Algorithm maintains A[low] $\leq$ key $\leq$ A[high].

Ex. Binary search for 47.

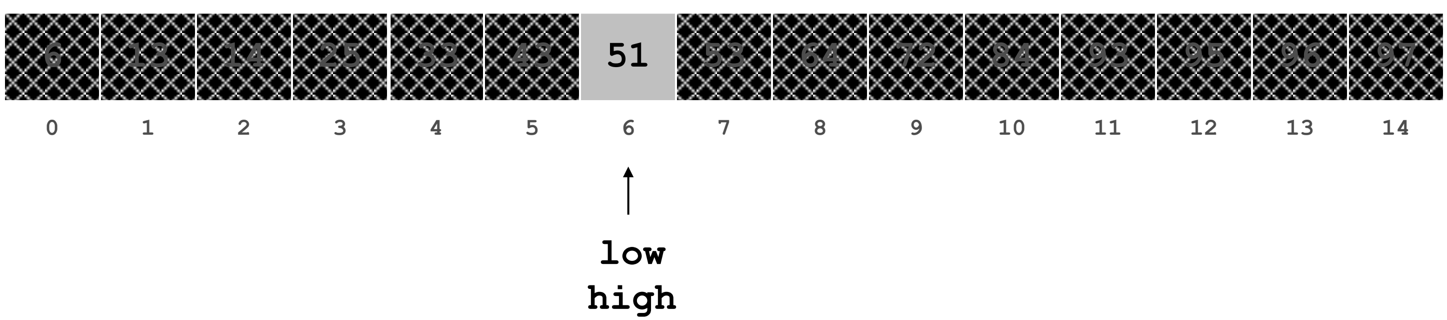


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` ≤ `key` ≤ `A[high]`.

Ex. Binary search for 47.

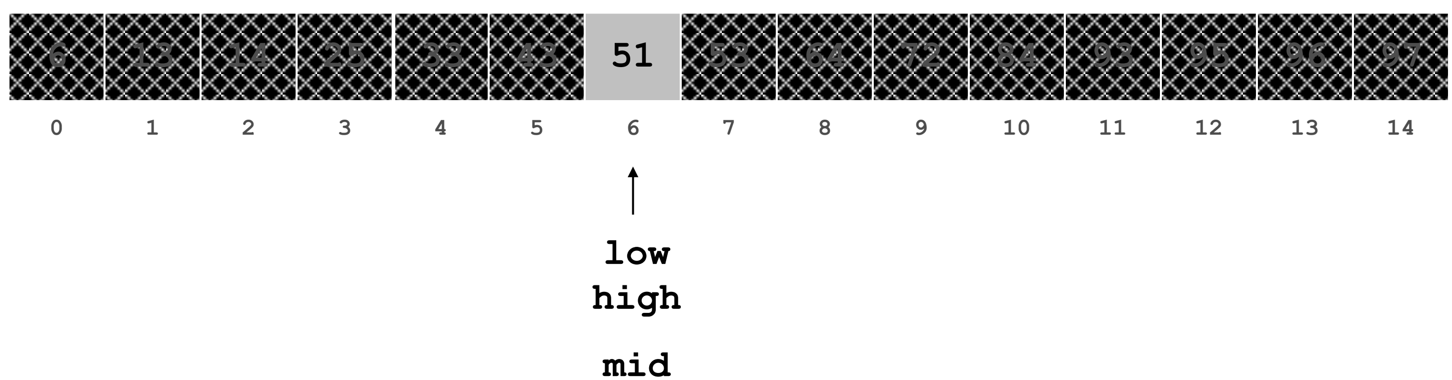


# Binary Search

Binary search. Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

Invariant. Algorithm maintains `A[low]` ≤ `key` ≤ `A[high]`.

Ex. Binary search for 47.

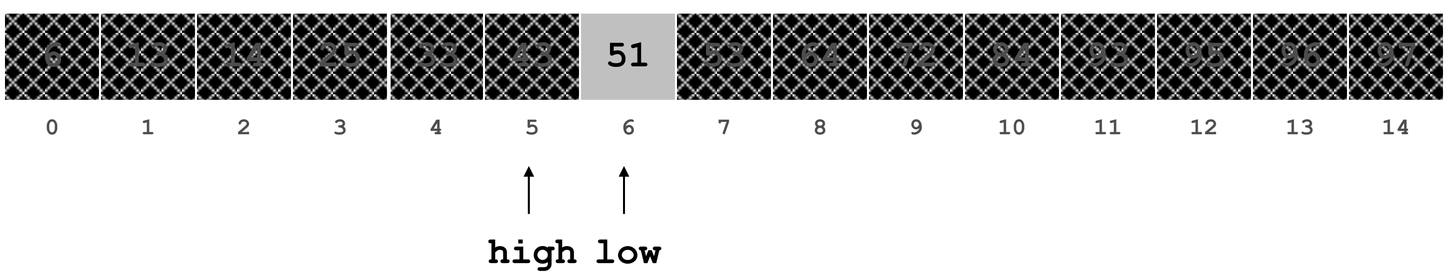


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains `A[low]` $\leq$ `key` $\leq$ `A[high]`.

**Ex.** Binary search for 47.




47 is not in the array

# Binary Search

Binary search running time

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$
$$\Longrightarrow T(n) \; = O(\log n)$$

Lesson: Additional structure (sorted array) can break the usual lower bound